

Cuadernillo Semestral de Actividades

Actualizado: 20 de septiembre de 2024

El presente cuadernillo posee un compilado con todos los ejercicios que se usarán durante el semestre en la asignatura. Los ejercicios están organizados en forma secuencial, siguiendo los contenidos que se van viendo en la materia.

Cada semana les indicaremos cuáles son los ejercicios en los que deberían enfocarse para estar al día y algunos de ellos serán discutidos en la explicación de práctica.

Recomendación importante:

Los contenidos de la materia se incorporan y fijan mejor cuando uno intenta aplicarlos - **no alcanza con ver un ejercicio resuelto por alguien más**. Para sacar el máximo provecho de los ejercicios, es importante que asistan a las consultas de práctica habiendo intentado resolverlos (tanto como les sea posible). De esa manera podrán hacer consultas más enfocadas y el docente podrá darles mejor feedback.

Ejercicio 1: WallPost

Primera parte

Se está construyendo una red social como Facebook o Twitter. Debemos definir una clase Wallpost con los siguientes atributos: un texto que se desea publicar, cantidad de likes ("me gusta") y una marca que indica si es destacado o no. La clase es subclase de Object.

Para realizar este ejercicio, utilice el recurso que se encuentra en el sitio de la cátedra (o que puede descargar desde [acá](#)). Para importar el proyecto, siga los pasos explicados en el documento "*Trabajando con proyectos Maven, importar un proyecto*". Allí verá que existe la interface Wallpost y la clase WallpostImpl que implementa la interfaz anterior. Una vez importado, dentro del mismo, debe completar la clase WallPostImpl para que entienda:

```
/*
 * Permite construir una instancia del WallpostImpl.
 * Luego de la invocación, debe tener como texto: "Undefined post",
 * no debe estar marcado como destacado y la cantidad de "Me gusta" debe ser 0.
 */
public WallPostImpl()
```

E implemente el protocolo definido en la interfaz Wallpost como se detalla a continuación

```
/*
 * Retorna el texto descriptivo de la publicación
 */
public String getText()

/*
 * Asigna el texto descriptivo de la publicación
 */
public void setText (String descriptionText)

/*
 * Retorna la cantidad de "me gusta"
 */
public int getLikes()

/*
 * Incrementa la cantidad de likes en uno.
 */
public void like()

/*
 * Decrementa la cantidad de likes en uno. Si ya es 0, no hace nada.
 */
public void dislike()

/*
 * Retorna true si el post está marcado como destacado, false en caso contrario
 */
public boolean isFeatured()

/*
 * Cambia el post del estado destacado a no destacado y viceversa.
 */
public void toggleFeatured()
```

Segunda parte

Utilice los tests provistos por la cátedra para comprobar que su implementación de Wallpost es correcta. Estos se encuentran en el mismo proyecto, en la carpeta test, clase WallPostTest.

Para ejecutar los tests simplemente haga click derecho sobre el proyecto y utilice la opción Run As >> JUnit Test. Al ejecutarlo, se abrirá una ventana con el resultado de la evaluación de los tests. Siéntase libre de investigar la implementación de la clase de test. Ya veremos en detalle cómo implementarlas.



En el informe, Runs indica la cantidad de test que se ejecutaron. En Errors se indica la cantidad que dieron error y en Failures se indica la cantidad que tuvieron alguna falla, es decir, los resultados no son los esperados. Abajo, se muestra el Failure Trace del test que falló. Si lo selecciona, mostrará el mensaje de error correspondiente a ese test, que le ayudará a encontrar la falla. Si hace click sobre alguno de los test, se abrirá su implementación en el editor.

Tercera parte

Una vez que su implementación pasa los tests de la primera parte puede utilizar la ventana que se muestra a continuación, la cual permite inspeccionar y manipular el post (definir su texto, hacer like / dislike y marcarlo como destacado).



Para visualizar la ventana, sobre el proyecto, usar la opción del menú contextual Run As >> Java Application. La ventana permite cambiar el texto del post, incrementar la cantidad de likes, etc. El botón Print to Console imprimirá los datos del post en la consola.

Ejercicio 2: Balanza Electrónica

En el taller de programación ud programó una balanza electrónica. Volveremos a programarla, con algún requerimiento adicional.

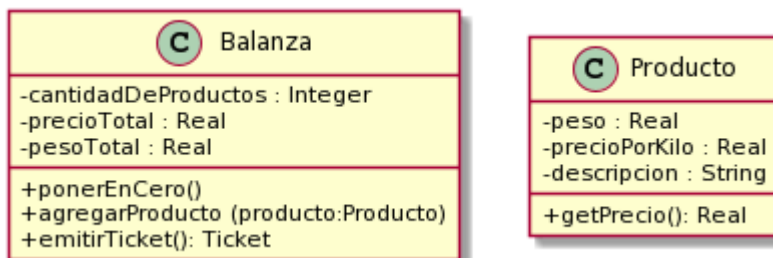
En términos generales, la Balanza electrónica recibe productos (uno a uno), y calcula dos totales: peso total y precio total. Además, la balanza puede poner en cero todos sus valores.

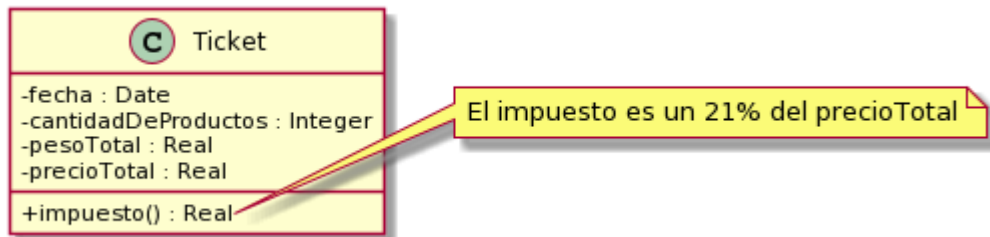
La balanza no guarda los productos. Luego emite un ticket que indica el número de productos considerados, peso total, precio total.

Tareas:

a) Implemente:

Cree un nuevo proyecto Maven llamado `balanzaElectronica`, siguiendo los pasos del documento “*Trabajando con proyectos Maven, crear un proyecto Maven nuevo*”. En el paquete correspondiente, programe las clases que se muestran a continuación.





Observe que no se documentan en el diagrama los mensajes que nos permiten obtener y establecer los atributos de los objetos (accessors). Aunque no los incluimos, verá que los tests fallan si no los implementa. Consulte con el ayudante para identificar, a partir de los tests que fallan, cuales son los accessors necesarios (pista: todos menos los setters de balanza).

Todas las clases son subclasses de Object.

Nota: Para las fechas, utilizaremos la clase `java.time.LocalDate`. Para crear la fecha actual, puede utilizar `LocalDate.now()`. También es posible crear fechas distintas a la actual. Puede investigar más sobre esta clase en

<https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html>

b) Probando su implementación:

Para realizar este ejercicio, utilice el recurso que se encuentra en el sitio de la cátedra o que puede descargar desde este [link](#). En este caso, se trata de dos clases, `BalanzaTest` y `ProductoTest`, las cuales debe agregar dentro del paquete tests. Haga las modificaciones necesarias para que el proyecto no tenga errores.

Si todo salió bien, su implementación debería pasar las pruebas que definen las clases agregadas en el paso anterior. El propósito de estas clases es ejercitar una instancia de la clase `Balanza` y verificar que se comporta correctamente.

Ejercicio 3: Presupuestos

Un presupuesto se utiliza para detallar los precios de un conjunto de productos que se desean adquirir. Se realiza para una fecha específica y es solicitado por un cliente, proporcionando una visión de los costos asociados.

El siguiente diagrama muestra un diseño para este dominio.



Tareas:

a) Implemente:

Defina el proyecto Ejercicio 3 - Presupuesto y dentro de él implemente las clases que se observan en el diagrama. Ambas son subclases de Object.

b) Discuta y reflexione

Preste atención a los siguientes aspectos:

- ¿Cuáles son las variables de instancia de cada clase?
- ¿Qué variables inicializa? ¿De qué formas se puede realizar esta inicialización?
- ¿Qué ventajas y desventajas encuentra en cada una de ellas?

c) Probando su código:

Utilice los [tests provistos](#) para confirmar que su implementación ofrece la funcionalidad esperada. En este caso, se trata de dos clases: ItemTest y PresupuestoTest, que debe agregar dentro del paquete tests. Haga las modificaciones necesarias para que el proyecto no tenga errores. Siéntase libre de explorar las clases de test para intentar entender qué es lo que hacen.

Ejercicio 4: Balanza mejorada

Realizando el ejercicio de los presupuestos, aprendimos que un objeto puede tener una colección de otros objetos. Con esto en mente, ahora queremos mejorar la balanza implementada en el ejercicio 2.

Tarea 1

Mejorar la balanza para que recuerde los productos ingresados (los mantenga en una colección). Analice de qué forma puede realizarse este nuevo requerimiento e implemente el mensaje

```
public List<Producto> getProductos()
```

que retorna todos los productos ingresados a la balanza (en la compra actual, es decir, desde la última vez que se la puso a cero).

¿Qué cambio produce este nuevo requerimiento en la implementación del mensaje *ponerEnCero()* ?

¿Es necesario, ahora, almacenar los totales en la balanza? ¿Se pueden obtener estos valores de otra forma?

Tarea 2

Con esta nueva funcionalidad, podemos enriquecer al Ticket, haciendo que él también conozca a los productos (a futuro podríamos imprimir el detalle). Ticket también debería entender el mensaje `public List<Producto> getProductos()`.

- ¿Qué cambios cree necesarios en Ticket para que pueda conocer a los productos?
- ¿Estos cambios modifican las responsabilidades ya asignadas de realizar cálculo del precio total?. ¿El ticket adquiere nuevas responsabilidades que antes no tenía?

Tarea 3

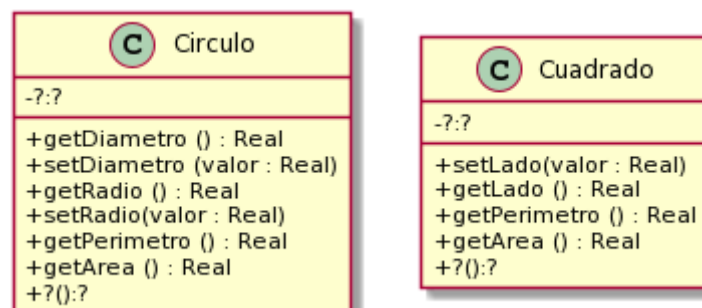
Después de hacer estos cambios, ¿siguen pasando los tests? ¿Está bien que sea así?

Ejercicio 5: Figuras y cuerpos

Figuras en 2D

En Taller de Programación definió clases para representar figuras geométricas. Retomaremos ese ejercicio para trabajar con Cuadrados y Círculos.

El siguiente diagrama de clases documenta los mensajes que estos objetos deben entender.



Fórmulas y mensajes útiles:

- Diámetro del círculo: $\text{radio} * 2$
- Perímetro del círculo: $\pi * \text{diámetro}$
- Área del círculo: $\pi * \text{radio}^2$

- π se obtiene enviando el mensaje #pi a la clase Float (Float pi) (ahora Math.PI)

Tareas:

a) Implementación:

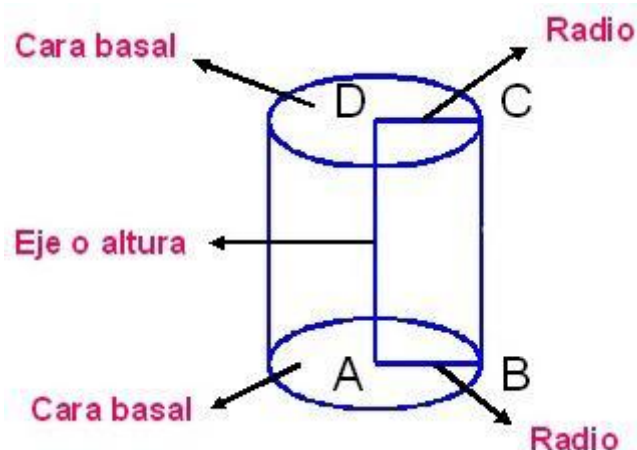
Defina un nuevo proyecto figurasYCuerpos. Implemente las clases Circulo y Cuadrado, siendo ambas subclases de Object. Decida usted qué variables de instancia son necesarias. Puede agregar mensajes adicionales si lo cree necesario.

b) Discuta y reflexione

¿Qué variables de instancia definió? ¿Pudo hacerlo de otra manera? ¿Qué ventajas encuentra en la forma en que lo realizó?

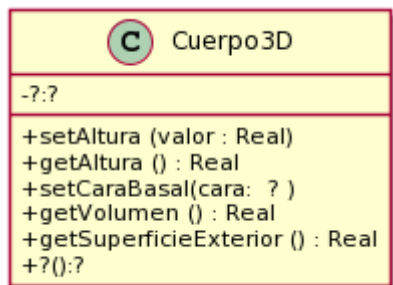
Cuerpos en 3D

Ahora que tenemos Círculos y Cuadrados, podemos usarlos para construir cuerpos (en 3D) y calcular su volumen y superficie o área exterior. Vamos a pensar a un cilindro como "un cuerpo que tiene una figura 2D como cara basal y que tiene una altura (vea la siguiente imagen)". Si en el lugar de la figura2D tuviera un círculo, se formaría el siguiente cuerpo 3D.



Si reemplazamos la cara basal por un rectángulo, tendremos un prisma (una caja de zapatos).

El siguiente diagrama de clases documenta los mensajes que entiende un cuerpo3D.



Fórmulas útiles:

- El área o superficie exterior de un cuerpo es:
 $2 * \text{área-cara-basal} + \text{perímetro-cara-basal} * \text{altura-del-cuerpo}$
- El volumen de un cuerpo es: $\text{área-cara-basal} * \text{altura}$

Más info interesante: A la figura que da forma al cuerpo (el círculo o el cuadrado en nuestro caso) se le llama directriz. Y a la recta en la que se mueve se llama generatriz. En [wikipedia \(Cilindro\)](https://es.wikipedia.org/wiki/Cilindro)¹ se puede aprender un poco más al respecto.

Tareas:

a) Implementación

Implemente la clase Cuerpo 3D, la cuál es subclase de Object. Decida usted qué variables de instancia son necesarias. También decida si es necesario hacer cambios en las figuras 2D.

b) Pruebas automatizadas

Siguiendo los ejemplos de ejercicios anteriores, ejecute [las pruebas automatizadas provistas](#). En este caso, se trata de tres clases (CuerpoTest, TestCirculo y TestCuadrado) que debe agregar dentro del paquete tests. Haga las modificaciones necesarias para que el proyecto no tenga errores. Si algún test no pasa, consulte al ayudante.

c) Discuta y reflexione

Discuta con el ayudante sus elecciones de variables de instancia y métodos adicionales. ¿Es necesario todo lo que definió?

Ejercicio 6: Genealogía salvaje

En una reserva de vida salvaje (como la estación de cría ECAS, en el camino Centenario), los cuidadores quieren llevar registro detallado de los animales que cuidan y sus familias. Para ello nos han pedido ayuda. Debemos:

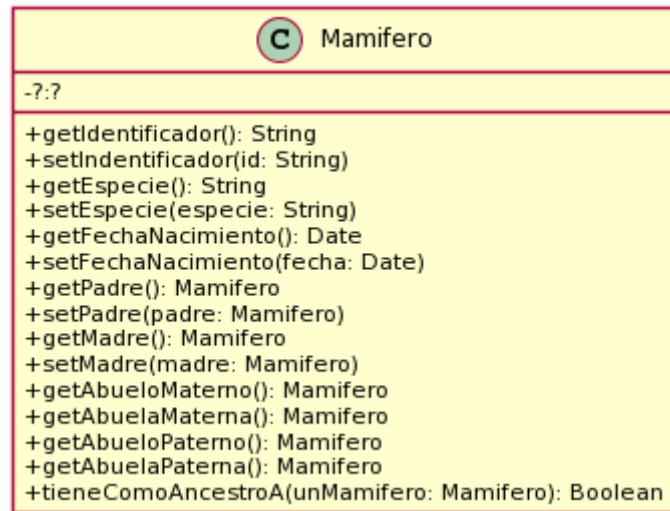
Tareas:

a) Complete el diseño e implemente

Modelar una solución en objetos e implementar la clase Mamífero (como subclase de Object). El siguiente diagrama de clases (incompleto) nos da una idea de los mensajes que un mamífero entiende.

Proponga una solución para el método *tieneComoAncestroA(...)* y *deje la implementación para el final y discuta su solución con el ayudante*.

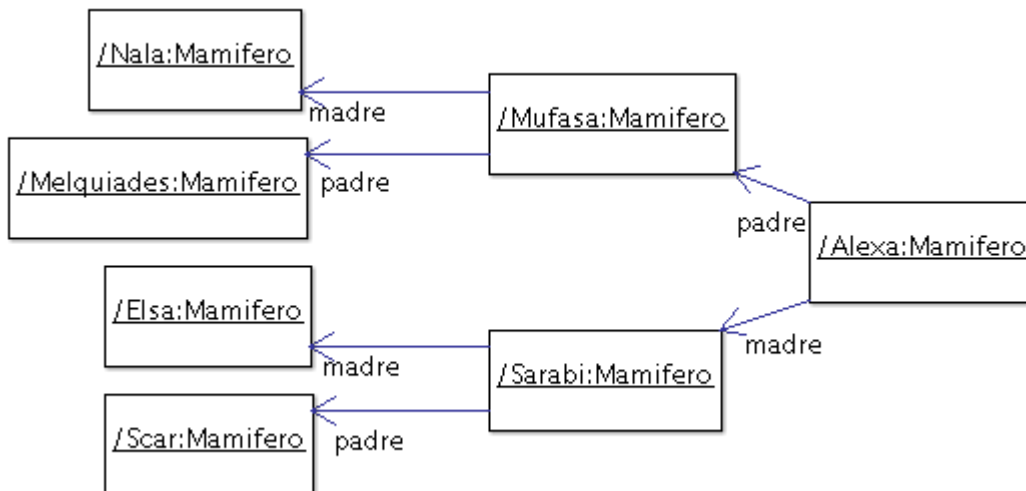
¹ <https://es.wikipedia.org/wiki/Cilindro>



Complete el diagrama de clases para reflejar los atributos y relaciones requeridas en su solución.

b) Pruebas automatizadas

Siguiendo los ejemplos de ejercicios anteriores, ejecute [las pruebas automatizadas provistas](#). En este caso, se trata de una clase, MamiferoTest, que debe agregar dentro del paquete tests. En esta clase se trabaja con la familia mostrada en la siguiente figura.



En el diagrama se puede apreciar el nombre/identificador de cada uno de ellos (por ejemplo Nala, Mufasa, Alexa, etc).

Haga las modificaciones necesarias para que el proyecto no tenga errores. Si algún test no pasa, consulte al ayudante.

Ejercicio 7: Red de Alumbrado

Imagine una red de alumbrado donde cada farola está conectada a una o varias vecinas formando un [grafo conexo](https://es.wikipedia.org/wiki/Grafo_conexo)². Cada una de las farolas tiene un interruptor. Es suficiente con encender o apagar una farola cualquiera para que se enciendan o apaguen todas las demás. Sin embargo, si se intenta apagar una farola apagada (o si se intenta encender una farola encendida) no habrá ningún efecto, ya que no se propagará esta acción hacia las vecinas.

La funcionalidad a proveer permite:

1. crear farolas (inicialmente están apagadas)
2. conectar farolas a tantas vecinas como uno quiera (las conexiones son bi-direccionales)
3. encender una farola (y obtener el efecto antes descrito)
4. apagar una farola (y obtener el efecto antes descrito)

Tareas:

a) Modele e implemente

1. Realice el diagrama UML de clases de la solución al problema.

2. Implemente en Java, la clase Farola, como subclase de Object, con los siguientes métodos:

```
/*
 * Crear una farola. Debe inicializarla como apagada
 */
public Farola ()

/*
 * Crea la relación de vecinos entre las farolas. La relación de vecinos
 * entre las farolas es recíproca, es decir el receptor del mensaje será vecino
 * de otraFarola, al igual que otraFarola también se convertirá en vecina del
 * receptor del mensaje
 */
public void pairWithNeighbor( Farola otraFarola )

/*
 * Retorna sus farolas vecinas
 */
public List<Farola> getNeighbors ()

/*
 * Si la farola no está encendida, la enciende y propaga la acción.
 */
public void turnOn()

/*
 * Si la farola no está apagada, la apaga y propaga la acción.
 */
public void turnOff()
```

² https://es.wikipedia.org/wiki/Grafo_conexo

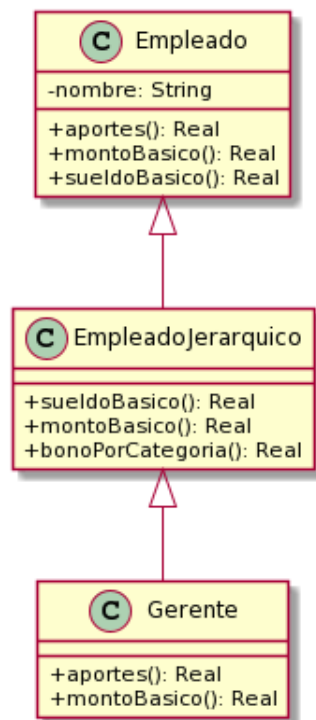
```
/*  
 * Retorna true si la farola está encendida.  
 */  
public boolean isOn()
```

b) Verifique su solución con las pruebas automatizadas

Utilice los [tests provistos](#) por la cátedra para probar las implementaciones del punto 2.

Ejercicio 8: Method lookup con Empleados

Sea la jerarquía de `Empleado` como muestra la figura de la izquierda, cuya implementación de referencia se incluye en la tabla de la derecha.



Empleado	EmpleadoJerarquico	Gerente
<pre>public double montoBasico() { return 35000; }</pre>	<pre>public double sueldoBasico() { return super.sueldoBasico()+ this.bonoPorCategoria(); }</pre>	<pre>public double aportes() { return this.montoBasico() * 0.05d; }</pre>
<pre>public double aportes(){ return 13500; }</pre>	<pre>public double montoBasico() { return 45000; }</pre>	<pre>public double montoBasico() { return 57000; }</pre>
<pre>public double sueldoBasico() { return this.montoBasico() + this.aportes(); }</pre>	<pre>public double bonoPorCategoria() { return 8000; }</pre>	

Analice cada uno de los siguientes fragmentos de código y resuelva las tareas indicadas abajo:

```
Gerente alan = new Gerente("Alan Turing");
double aportesDeAlan = alan.aportes();
```

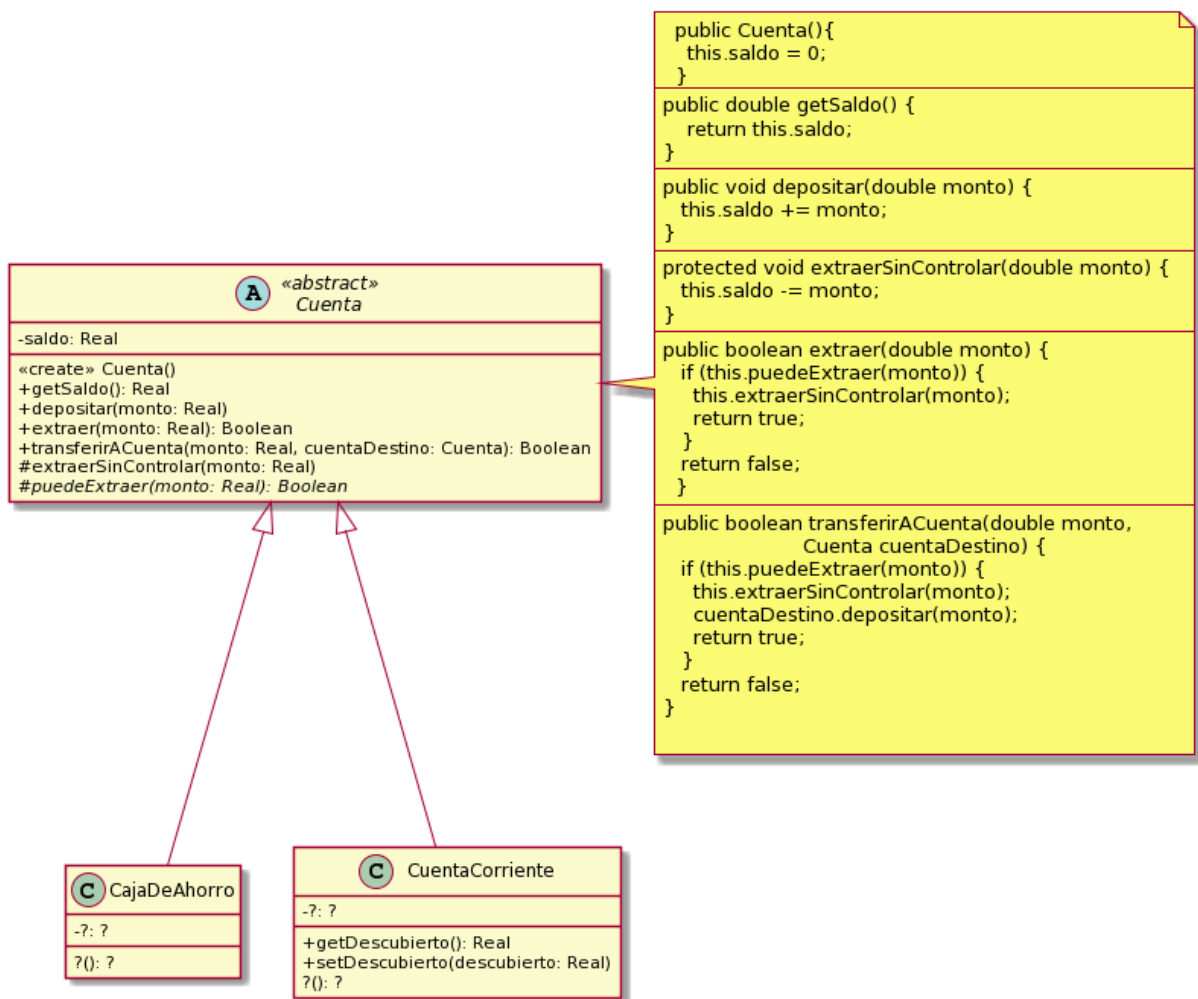
```
Gerente alan = new Gerente("Alan Turing");
double sueldoBasicoDeAlan = alan.sueldoBasico();
```

Tareas:

1. Liste todos los métodos, indicando nombre y clase, que son ejecutados como resultado del envío del último mensaje de cada fragmento de código (por ejemplo, (1) método +aportes de la clase Empleado, (2) ...)
2. ¿Qué valores tendrán las variables aportesDeAlan y sueldoBasicoDeAlan luego de ejecutar cada fragmento de código?

Ejercicio 9: Cuenta con ganchos

Observe con detenimiento el diseño que se muestra en el siguiente diagrama. La clase cuenta es *abstracta*. El método `puedeExtraer()` es abstracto. Las clases `CajaDeAhorro` y `CuentaCorriente` son concretas y están incompletas.



Tarea A: Complete la implementación de las clases `CajaDeAhorro` y `CuentaCorriente` para que se puedan efectuar depósitos, extracciones y transferencias teniendo en cuenta los siguientes criterios.

- 1) Las **cajas de ahorro** solo pueden extraer y transferir cuando cuentan con fondos suficientes.
- 2) Las extracciones, los depósitos y las transferencias desde **cajas de ahorro** tienen un costo adicional de 2% del monto en cuestión (téngalo en cuenta antes de permitir una extracción o transferencia desde caja de ahorro).
- 3) Las **cuentas corrientes** pueden extraer aún cuando el saldo de la cuenta sea insuficiente. Sin embargo, no deben superar cierto límite por debajo del saldo. Dicho límite se conoce como límite de descubierto (algo así como el máximo saldo negativo permitido). Ese límite es diferente para cada cuenta (lo negocia el cliente con la gente del banco).
- 4) Cuando se abre una **cuenta corriente**, su límite descubierto es 0 (no olvide definir el constructor por default).

Tarea B: Reflexione, charle con el ayudante y responda a las siguientes preguntas.

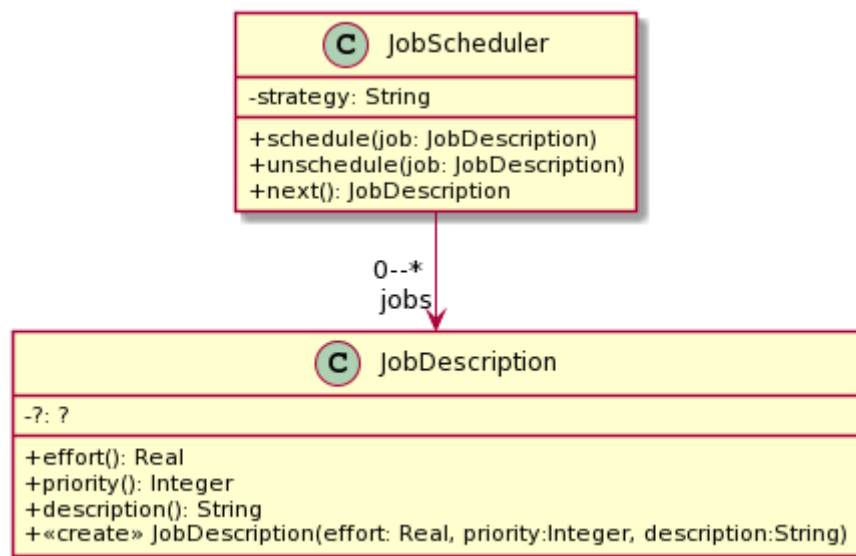
- a) ¿Por qué cree que este ejercicio se llama "Cuenta con ganchos"?

- En las implementaciones de los métodos `extraer()` y `transferirACuenta()` que se ven en el diagrama, ¿quién es `this`? ¿Puede decir de qué clase es `this`?
- ¿Por qué decidimos que los métodos `puedeExtraer()` y `extraerSinControlar` tengan visibilidad "protegido"?
- ¿Se puede transferir de una caja de ahorro a una cuenta corriente y viceversa? ¿por qué? ¡Pruébalo!
- ¿Cómo se declara en Java un método abstracto? ¿Es obligatorio implementarlo? ¿Qué dice el compilador de Java si una subclase no implementa un método abstracto que hereda?

Tarea C: Escriba los tests de unidad que crea necesarios para validar que su implementación funciona adecuadamente.

Ejercicio 10: Job Scheduler

El `JobScheduler` es un objeto cuya responsabilidad es determinar qué trabajo debe resolverse a continuación. El siguiente diseño ayuda a entender cómo funciona la implementación actual del `JobScheduler`.



- El mensaje `schedule(job: JobDescription)` recibe un job (trabajo) y lo agrega al final de la colección de trabajos pendientes.
- El mensaje `next()` determina cuál es el siguiente trabajo de la colección que debe ser atendido, lo retorna, y lo quita de la colección.

En la implementación actual del método `next()`, el `JobScheduler` utiliza el valor de la variable `strategy` para determinar cómo elegir el siguiente trabajo.

Dicha implementación presenta dos serios problemas de diseño:

- Secuencia de ifs (o sentencia switch/case) para implementar alternativas de un mismo comportamiento.
- Código duplicado.

Tareas:

a) **Analice el código existente**

Utilice el [código y los tests](#) provistos por la cátedra y aplique lo aprendido (en particular en relación a herencia y polimorfismo) para eliminar los problemas mencionados. Siéntase libre de agregar nuevas clases como considere necesario. También puede cambiar la forma en la que los objetos se crean e inicializan. Asuma que una vez elegida una estrategia para un scheduler no puede cambiarse.

b) **Verifique su solución con las pruebas automatizadas**

Sus cambios probablemente hagan que los tests dejen de funcionar. Corríjalos y mejórellos como sea necesario.

Ejercicio 11: El Inversor

Estamos desarrollando una aplicación móvil para que un inversor pueda conocer el estado de sus inversiones. El sistema permite manejar dos tipos de inversiones: Inversión en acciones e inversión en plazo fijo. Nuestro sistema representa al inversor y a cada uno de los tipos de inversiones con una clase.

- La clase `InversionEnAcciones` tiene las siguientes variables de instancia:

```
String nombre;  
int cantidad;  
double valorUnitario;
```

- La clase `PlazoFijo` tiene las siguientes variables de instancia:

```
LocalDate fechaDeConstitucion;  
double montoDepositado;  
double porcentajeDelInteresDiario;
```

- La clase `Inversor` tiene las siguientes variables de instancia:

```
String nombre;  
List<?> inversiones;
```

La variable `inversiones` de la clase `Inversor` es una colección con instancias de cualquiera de las dos clases de inversiones que pueden estar mezcladas.

Cuando se quiere saber cuánto dinero representan las inversiones del inversor, se envía al mismo el mensaje `valorActual()`.

Tareas:

a) **Modele e implemente**

- 1) Realice el diagrama UML de clases de la solución al problema.
- 2) Implemente en Java lo que considere necesario para que las instancias de `Inversor` entiendan el mensaje `valorActual()` teniendo en cuenta los siguientes criterios:

- El valor actual de las inversiones de un inversor es la suma de los valores actuales de cada una de las inversiones en su cartera (su colección de inversiones).
- El valor actual de un **PlazoFijo** equivale al *montoDepositado* incrementado como corresponda por el porcentaje de interés diario, desde la fecha de constitución a la fecha actual (la del momento en el que se hace el cálculo).
- El valor actual de una **InversionEnAcciones** se calcula multiplicando el número de acciones por el valor unitario de las mismas.
- Recordatorio: No olvide la inicialización.

b) Pruebas automatizadas

Implemente los tests (JUnit) que considere necesarios.